
# COVER SHEET

# A process-based methodology for designing event-based mobile composite applications

Tore Fjellheim Stephen Milliner Marlon Dumas

*Queensland University of Technology, Faculty of Information Technology, Brisbane, Australia*

Julien Vayssière

*SAP Research, Brisbane, Australia*

**Abstract**

Mobile application developers should be able to specify how applications can adapt to changing conditions, and to later reconfigure the application to suit new circumstances. Event-based communication have been advocated to facilitate such dynamic changes. Event-based models, however, are fragmented, which makes it difficult to understand the dependencies between components. A process-oriented methodology overcomes this issue, by specifying dependencies according to a process model. This paper describes a methodology that combines the comprehensibility and manageability of control from process-oriented methodologies, with the flexibility of event-based communication. This enables fine-grained adaptation of process-oriented applications.

*Key words:* Mobile, Methodology, Process-Oriented, Event-Based, Coordination

## 1 Introduction

Mobile environments are typically characterised by limited device resources, potential disconnections, and highly dynamic context. Composite applications, consisting of several autonomous components, have been proposed as a solution to some of these issues in mobile computing [13]. Developing composite

*Email addresses:* t.fjellheim@qut.edu.au (Tore Fjellheim), s.milliner@qut.edu.au (Stephen Milliner), m.dumas@qut.edu.au (Marlon Dumas), julien.vayssiere@sap.com (Julien Vayssière).

applications, requires that developers specify how these components should be orchestrated, most notably their dependencies in terms of flow of control and data. Process-based modeling presents developers with a method for specifying this. The major advantage of using a process-oriented approach for application development is that it provides an easy-to-comprehend and global view of the dependencies between the underlying applications and components. However, in existing process-oriented systems these dependencies have to be completely specified before deployment [1]. In mobile environments however, changes occur frequently and exceptions are numerous.

Mobile application developers should therefore be able to specify, during application design, how adaptation occurs. This requires that programmers cater for mobile computing issues during development, including considerations such as tradeoffs between resource usage, application performance and user satisfaction. Application developers must therefore consider aspects of personalising applications to suit the requirements or preferences of specific users, and adapting the behaviour of composite applications based on the users' context (e.g. location, device or network connection). Thus, the developer should be able to specify a set of process models which describe how various aspects of the application adapt to changing conditions.

However, an approach where the designer specifies all possible paths in the process model is impractical and leads to models that are large and unintelligible. Applications operating in dynamic mobile environments may be best served by a "just-in-time" approach, where adaptation and personalisation may be done after the process has been deployed and without requiring all executions to perfectly align with the original process model. To support a just-in-time approach, we advocate an event-based coordination approach to execute process-oriented composite applications. Due to its finer-grained nature, event-based coordination approaches have several advantages over process-based ones when it comes to runtime adaptation and reconfiguration [12]. By translating process models of composite applications into event-based models and using the latter in the runtime environment, it becomes possible, by adding and removing event-based rules (e.g. event subscriptions related to a specific task), to overlay behaviour on top of already deployed composite applications in response to special requirements or unforeseen situations.

To facilitate runtime adaptation, such composite applications should use decoupled interactions and support dynamic deployment [6]. Hence, a composite application will comprise several autonomous communicating components, interacting via a shared memory space to achieve the correct functionality. Certain components can be migrated to the mobile device, or run remotely on a server, as circumstances or users dictate. The developer must therefore not only be able to specify the interactions between the components, but also aspects such as how remote accesses (distribution of data), or migration of

behaviour (i.e. code) between devices are handled. Previous work in mobility has focused on architectures, context handling, user interface development, or algorithms for transcoding data for delivery to mobile devices. Little focus has been placed on methodologies for specifying interactions, distribution of data or movement of behaviour.

The main contribution of this paper is a methodology whereby developers can create adaptive user-centric applications. We present developers with heuristics and guidelines for how to specify the interactions, data distribution and component migration. All these aspects are specified using a mainstream process modeling notation (UML Activity Diagrams 2.0). These process models can be translated into an event-based model described through coordination rules made up of composite event specifications, predicates, and a small number of publishing/sharing primitives. The resulting event-based model can be executed on top of a shared object Space infrastructure and personalisation and unanticipated adaptation is achieved by adding rules (encoded as active objects) into the shared Space.

The paper is structured as follows. First, we describe the infrastructure and coordination primitives upon which our proposal relies (Section 2). Then we outline the methodology, explaining how the developer can specify aspects of a mobile application using UML Activity Diagrams (Section 3, using a mobile process use-case). In Section 4 we introduce a technique for translating a process model captured as a UML activity diagram into an event-based coordination model, and discuss how adaptation can be achieved by adding, enabling and disabling rules in the event-based model. Finally, we discuss related work (Section 5) and conclude (Section 6).

## 2 Infrastructure for event-based model execution

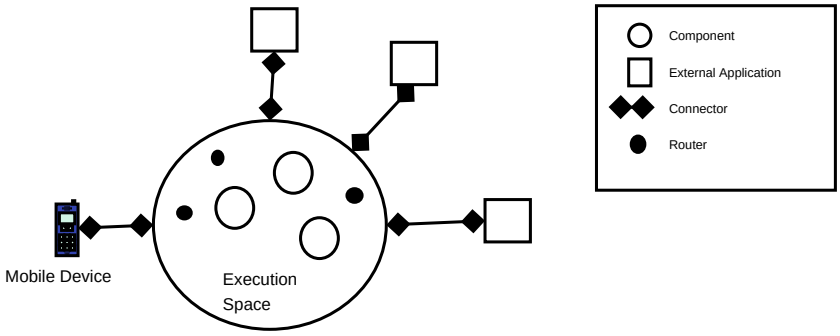


Fig. 1. System Overview

Event-based mobile applications use event messages to communicate between application components. For the execution of event-based mobile applications, we use the 3DMA architecture (Decomposed, Distributed, Decoupled Mobile

Applications) [6]. Figure 1 shows the infrastructure overview consisting of a main execution Space, and several external applications. The 3DMA architecture focuses on three principles which outline requirements for how mobile applications and architectures should be built in order to facilitate adaptation to dynamic changes in the environment. These are 1) Decomposition, 2) Distribution and 3) Decoupling.

Decomposition means that the architecture must support component-based development and execution, as mobile applications will be component based to facilitate fine grained adaptation. Distribution refers to the support for runtime movement of data and behaviour between devices and servers. Runtime distribution of application activities, allows clients to vary between "thick"-client and "thin"-client configurations depending on network or device constraints. Finally, decoupling means that application components communicate anonymously via the environment and not directly. Decoupled interactions makes swapping components and changing interactions easier.

To support decoupled interactions between components, and to enable execution of the event-based coordination models that will be derived from process models, we require an execution infrastructure with support for: 1) event publishing, data transfer/sharing, and complex event subscription; 2) association of reactions to event occurrences; and 3) runtime re-configuration so that new event subscriptions and reaction rules can be added anytime. The 3DMA architecture builds on the *Active Object Space* (AOS) [5,6] to achieve support for these requirements. The AOS runs both on the server side, and on the mobile device to facilitate process execution on both locations.

### *2.1 Active Object Spaces*

The AOS is a type of communication infrastructure known as *coordination middleware* which has its roots in the tuple space model underlying the Linda system [8]. At the centre of the AOS is a shared memory (the *space*). The AOS uses this to support undirected decoupled communication based on four elementary operations, namely *read*, *write*, *take* and *notify*. A read operation copies an object from the Space matching a given object template; a take operation moves an object matching a given object template out of the Space; a write operation puts an object on the Space; and a notify operation registers a subscription for a composite event expressed as a set of object templates. Whenever there is a combination of objects present in the Space that matches these object templates, a notification will be sent to the subscriber application. An *object template* is an expression composed of a class name and a set of equality constraints on the properties of that class. An object matches a template if its class is equal to or is a sub-class of the class designated by the

template and fulfills the template's constraints.

An originality of the AOS with respect to other object-oriented coordination middleware lies in its support for *active objects* (AO), that is, objects with their own thread of control that run on the Space. Active objects can read and write passive objects to/from the Space, subscribe to events, and receive notifications from the Space. Active objects operating on a shared memory and writing and taking objects to/from this Space, constitutes a powerful paradigm not only for executing event-based coordination models, but also for re-configuring these models after their deployment. Active objects can be deployed, suspended, resumed, and destroyed by applications running outside the Space, or by other active objects, at any time.

The AOS architecture provides a foundation, upon which explicit support for execution of event-based mobile applications can be built. In order to build applications upon this foundation, we introduce the concept of coordinators (special AOs). Coordinators interact with each other and other application components to provide explicit support for executing event-based applications, including their interactions, and the distribution of application data and behaviour.

### *2.2 Coordinators*

A *coordinator* is an active object that is deployed in the Space to coordinate work (e.g. to perform synchronization or data transfer) and operates in a loop until suspended or destroyed. Its basic task is to react to the appearance of specific objects in the Space, and to create new objects based on these where warranted. Coordinators thus embody the two notions of events and process control. There are two types of coordinators: 1) connectors and 2) routers.

#### *2.2.1 Connectors*

A *connector* is a type of coordinator dedicated to enabling a connection between the Space and one or several external applications or services. Connectors "wrap" external applications which rely on other communication protocols and interfaces, making interaction with external applications appear identical to interaction with local components. Connectors interact with remote services and mobile devices and may implement two special types of tasks required for explicit movement of data or behaviour, *pull* and *push*. A pull operation results in the pull of data or behaviour from a remote Space to the connector's local Space (i.e. to where the connector is executing). A push operation results in the sending of data or behaviour from the local execution Space to a remote execution Space. As these operations may move instances

of applications, they provide the opportunity to move application state, for example, for the purpose of session migration [16].

### *2.2.2 Router*

Routers specify how and when external applications (through connectors) and application components execute. The routers thereby coordinate the interaction between these entities execute in order to facilitate the execution of instances of a process. A router is described by the following elements:

- Input set: A set made up of a combination of object templates and boolean conditions.
- Output: A set of expressions, each of which evaluates to an object.
- Stop set: A set containing a combination of object templates and boolean conditions.
- Replace set: A set of coordinators.

Routers are used as follows. Upon creation, the router will place a subscription with the Space for the set of object templates contained in its input set (i.e. the set obtained after removing the boolean conditions from the input set). Subsequently, the router will be notified whenever a set of objects matching these templates are available on the Space. At this point, the router evaluates the set of conditions in its input set. If all these conditions are true, the router proceeds to "read" the set of objects in question and to evaluate the transformation functions (i.e. the expressions in the "Output") with these objects as input. The objects resulting from the transformation are then written back to the Space. The "input set"thus captures the events and conditions that lead to the activation of a router (where an event corresponds to the arrival of an object to the Space). The "Output" on the other hand encodes the objects that the router will produce upon activation, i.e. such objects can be consumed by other coordinators, or by services or application components. Finally, if a set of objects matching the object templates in the stop set is found on the Space, the router will terminate its execution and replace itself by the set of routers specified in the replace set.

## *2.3 Components*

Component Active Objects perform application processing. Unlike external applications, components are typically not shared between different applications. The use of active objects to implement components provides autonomy and decoupling, facilitating changing of interactions depending on context.

Components and coordinators interact with each other via writing and reading

objects to the Space. Hence, creation of objects in the Space is performed by coordinators, components, and by other external applications. To ensure routers act upon objects related to the same instance of a process, each object in the Space contains a process instance identifier (*piid*). It is the responsibility of the components and coordinators to ensure that this piid is maintained. The output written to the Space should contain the same piid as the input data read. When a process completes, all objects with the corresponding piid are removed from the execution Space.

Objects in the Space can be used for two purposes. Firstly, objects facilitate the flow of data objects between components or from one application to another. Secondly, they can be used for signposting, indicating that a given step of work has been completed or that a given step of work is enabled but has not yet started. During execution, routers read and take objects denoting the completion (or failure) of tasks (i.e. *task completion objects*) and write into the Space objects denoting the enabling of tasks (i.e. *task enabling objects*). Components and connectors can read or take task enabling objects, execute the corresponding task by interacting with external applications, and may eventually write back task completion objects, which are then read by routers.

Context and preferences of users required for the execution of the application is assumed to be gathered through external sensors connected to a context service. This service provides context and preferences to the AOS as objects which can be stored in the active object space and used for process execution.

### *2.4 Interaction and Deployment Protocols*

Interaction between connectors and components follow a protocol similar to the execution protocol. Hence we can express movement of data and behavior in a similar manner to interaction. Remote distribution (sending of data and/or behaviour) thereby only becomes a special case of interaction. Such similar treatment of interaction, data distribution and deployment allows the creation of a unified model used to specify all aspects. The protocols for interaction and deployment/distribution both follow an event-based style, and can be described in terms of five elements namely: 1) Activation, 2) Input, 3) Processing, 4) Output and 5) Destination.

Interaction or deployment will be *Activated* when a router reacts to a set of input templates which specify an enablement condition for a component or a service. Based on this reaction, the router will produce an enablement message which is sent to the Space. Within this enablement message, there is a set of templates which describe what the component or service should accept as *Input* and perform *Processing* on. The component will use these templates to

read data from the Space and perform processing on it. Upon completion the component may produce some *Output* data, replacing the old data in case it is modified, and a signposting element to notify its completion. In certain cases, it may be necessary to specify what processing should occur next, in which case a router will read the output data and attach metadata to it describing its *Destination*. To enable component migration, behaviour components can be treated as data, and read as input into connectors.

The AOS does not provide a replace operation required for this interaction protocol and replacement is performed by a taken followed by a write. This operation should be made atomic in case of failures. Such transactional "all or nothing" behaviour can be implemented by dedicated type of active object. In addition, the AOS is responsible for persisting the objects on the space to facilitate recovery.

When a connector to a remote execution Space is activated through an enablement message, it will read the required input components and send them to the remote site it represents. How the input data to be transmitted is specified, is outlined in Section 3.5. In case of a failure during transmission, the connector will output the data which has not been transmitted, and allow another connector to continue the transfer. This is accompanied by a task completion or a task failure object. Connectors are thereby responsible for implementing fault tolerance mechanisms to network failures.

This section has outlined the architecture used for event-based process execution. In the following section we present an overview of the development process to design an event-based mobile application using process-oriented modeling.

## 3 Application design

The purpose of the methodology is to provide general guiding principles to be used when developing mobile applications. The methodology also aims at reducing the effort required for developers to use the architecture, thereby facilitating the creation of adaptive applications. During application design, a set of processes are created, each of these processes represent either the interaction between components, the migration of these components or the distribution of data. After the process design is complete, the process specifications are transformed into a set of routers, based on an algorithm. These routers are then deployed in the shared memory Space for execution.

Execution of these processes is often split between the mobile device Space and the server Space. The mobile device initially contains an AOS with an applica-

tion selector GUI component. Upon selecting a component for installation and execution, required processes (represented as routers), and a set of components are installed. During execution, when an interaction fails because interacting components are on different sites, either the data distribution aspect can be used to send the data to the remote site, or an activity (component) can be deployed to the device to perform the processing locally. When an activity is migrated to the mobile device, using the deployment aspect, its pre- and post-condition routers are also migrated, thereby moving parts of the process to the remote site.

The methodology consists of two main parts: firstly a set of guidelines outlining the design dependencies, to be considered during application development, and secondly a set of process-models, each describing a certain aspect of the application. In this paper we use UML Activity Diagrams for process modeling. UML activity diagrams are a widely used modeling notation and its constructs are representative of those found in other process modeling and process execution languages (e.g. sequence, fork, join, decision and merge nodes). Thus the proposed techniques can be adapted to other languages that rely on these constructs. Moreover, a recent study shows that UML activity diagrams (version 2.0) provide direct support for many common workflow patterns [18].

### *3.1 Design Dependencies*

Mobile application aspects can be divided into two parts, the environment model and the execution model. The environment model consists of a specification of the entities in the system (e.g. devices and users), as well as their context (e.g. location, and bandwidth), and their preferences and policies. This model provides the designer with the knowledge of when to adapt, thus, changes in the environment model are the causes of adaptation.

The environmental model consists of a set of variables which are referred to in the execution model in order to specify adaptation. In this paper, we take a simplified view of the environment model, only considering simple queries towards context or preferences, however, more advanced models (e.g. AWQL [11]) could potentially replace these queries.

The execution model consists of various static and dynamic aspects of the application. The dynamic aspects contain a specification of how the application adapts with reference to the environment model. The execution model thereby provides the effect of adaptation. The aspects of the execution model are derived from the architectural requirements of decomposition, distribution and decoupling. This entails that application developers must create components, specify how they interact and how data and behaviour is distributed. The

execution model consists of four parts in total, 1) the component activities, 2) the interactions, 3) the deployment, and 4) the data distribution. In this paper we focus on how process-oriented design, can specify how the dynamic aspects of interaction, deployment and distribution change according to the environment. Each dynamic aspect is further decomposed into sub-aspects, based on the five elements used to describe the protocol for event-based execution (Section 2.4), activation, input, processing, output and destination. Activation is used for control flow, whereas input, output and destination refers to data-flow. Processing is used for both control and data.

Mobile application developers must consider dependencies and impacts between application aspects. This impact analysis is a core aspect of our methodology, and provides the developer with guidelines and suggestions for how to design the application. A comprehensive list of the impacts is not possible in this paper due to limited space, but examples are provided. The guidelines firstly ensure that the application properly considers all impacts from environmental constraints. These impacts often engender some tradeoffs, and require certain design decisions to be made as to how each aspect should handle them. The guidelines also ensure that impacts from the various aspects of the execution model upon other aspects are handled. Such impacts arise when a design decision made in one aspect needs to be reflected in other aspects. In addition the developer should avoid situations where the effect of changes to the execution model, becomes a cause for further adaptation, causing a potentially infinite series of adaptation steps to take place. The consideration of impacts gives rise to design decisions which prevent such problems.

The dependencies developers must consider can be described as either an impact upon the environment or an effect upon other aspects. To guide developers through the design process, they must consider individual combinations of a sub-aspect and an environment variable. For each such combination a suggestion can be made to change the selected sub-aspect in order to positively influence the environment variable. Implementing such a suggestion may then positively or negatively impact upon other parts of the environment and may require additional tasks in implementing other application aspects. In the data distribution aspect, changing the activation sub-aspect would allow developers to change when data is distributed, changing the input or output would change what data is delivered, and changing the destination would alter where the data is delivered (e.g. multi channel in case of a disconnection). For example, to reduce the load on bandwidth (positive effect), the developer could be provided with a suggestion to use data compression (change the input). If this compression is a lossy type of compression (does not require decompression), then this would also positively affect the device memory because less data is delivered. If this type of compression is non-lossy, then a decompression component must be installed on the device. This would negatively impact upon the CPU load on the device, and would also require developers to specify in the

deployment aspect when this decompression component would be installed.

The next sections will discuss each of the four main aspects, and show some impacts from the environment model and how the execution model affect them. The aspects are discussed in relation to a scenario which is an example of a *mobile and personal workflow* [10], a process aimed at assisting a mobile user in the achievement of a goal that requires the execution of a number of tasks. Mobile and personal workflows constitute a class of process-oriented composite applications in which personalization and runtime adaptation are prominent requirements.

### *3.2 User and Application Activities*

The initial aspect to be specified is the activities in the application. In our scenario, a user is on a trip to attend a meeting. Before the meeting commences she runs a process-oriented application so that it assists her in the lead-up to the meeting. The user needs to have several tasks performed in some way to assist her in arriving at the meeting and having her notes available on her mobile device. The user will have a choice between catching a train or a taxi. In addition she must download the slides/notes that she requires for the meeting. The choice between the train or taxi will depend on many circumstances, and in addition, constraints such as disconnection will have impacts upon how and when the notes are downloaded. The user activities in our example are therefore: 1) Attend presentation, 2) Catch Train and 3) Catch Taxi. The developer must then consider what application activities are required to support these tasks, taking into account tasks used for displaying data, accessing data, and background processing. After this has been done, the developer specifies the interactions between these activities as shown in the next subsection.

### *3.3 Specifying Interactions*

The interaction aspect is the first of the dynamic aspects that is specified. The interaction aspect should be developed as platform independent as possible, and should assume that the model runs on a single device without need for deployment or data distribution. This is to make the same process (specification) reusable across various platforms. Device resources such as networks, platforms etc, should as far as possible not be included in the model, and the interaction aspect should rely only on user context, preferences or results of service executions. This reduces the impacts upon interactions, making this aspect easier to specify. The specification of data distribution or deployment

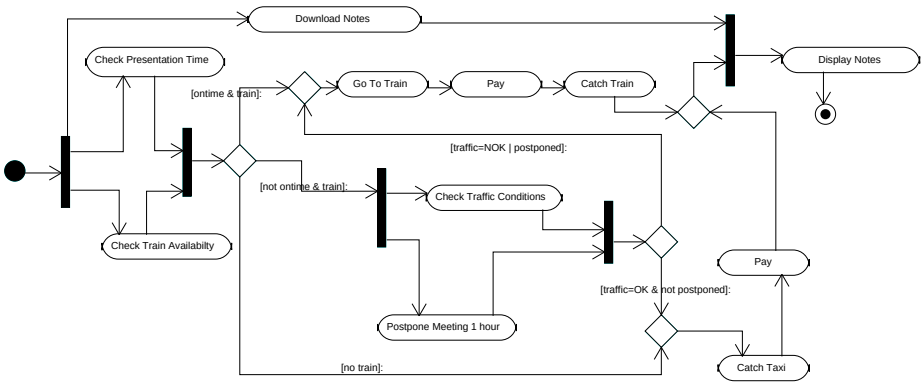


Fig. 2. Activity Diagram illustrating activity interactions

should be left to the appropriate aspects, to further reduce impacts upon this aspect.

Figure 2 shows the application activities in our case example, and the interactions between them. The process starts with three activities in parallel: 1) checking the presentation time, 2) checking the availability of trains to the destination, and 3) downloading meeting notes to the user's device (which may take some time due to low bandwidth). After the presentation time and the train availability of the train have been checked, three options are available: 1) If the user is "on time" AND "there is a train" that would take the user near the meeting's location, the user is directed to the train station; 2) If there is "no train", a taxi is automatically ordered; 3) If the user is "late" AND "there is a train", two new activities are started to determine if a taxi or a train is the best option for the user. At this point, the process checks the traffic conditions and tries to postpone the meeting by one hour (both actions in parallel). If the traffic is adverse, there is no point in catching a taxi, and the application will advise the user to catch the train. The same applies if the meeting is postponed. If however, there is favorable traffic and the meeting can not be postponed, the user will catch a taxi to get there sooner. Each transportation requires a payment. Payment is an activity which allows users to fill in payment details and send these details to their finance department to arrange for a refund (both of these steps are modeled as a single task "pay"). Finally, once the user is on her way to the meeting and the meeting notes have been downloaded, the application displays the notes.

In the interactions in this example we focus on the control-flow aspects of the interaction. We therefore ignore the input, output and destination sub-aspects, as these are related to data-flow. Each action in this activity diagram, has an associated activation condition, given by control flow and guard expressions. This condition specifies when the activity is executed. Guard expressions use variables which represent either: context data, such as user context or application context, or the output of a component or service. In the latter case, the developer must determine activities which produce data required for such

guard expressions. In case of context data, such data is automatically produced by an external context service (external application).

When specifying the activation (guard expressions) of actions, the developer must consider the impacts from the environment and other aspects. Interactions are specified independently from distribution of components and data, and so, these aspects do not impact upon the application interactions. The impacts from the environment entail that the developer must consider whether to allow activation of components to be based on user context or preferences. In our example, the final action is started when the user has caught a train or taxi, and the notes are ready. Alternatively it could be started when the user is at the meeting location.

### *3.4 Deployment*

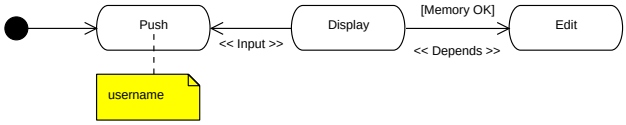


Fig. 3. Deployment Activity Diagram

Deployment is the process of moving an activity and its specification to a device or server for execution at a different location. A deployment process is separated in design from interaction, but executes at the same time as the main interaction process to determine when and what components move. The development aspect is created by considering for each activity which can potentially run on the device, when (activation) and to where they are deployed (destination). The process is specified using the connector activities "push" or "pull", which either send activities remotely or request activities for local deployment respectively.

When determining what is deployed, the initial activity to be moved, is linked to the pull/push activity using an input stereotype arrow. Upon activation of the pull/push activity, the input activity is migrated. The developer may also specify what dependencies this activity has. Dependencies specify what other activities are required, and when they are required. Each dependency is specified using a stereotype flow, called "depends". Guard expressions on the arrow specify under what conditions the activity is deployed.

The deployment process in Figure 3 is intended to execute on the mobile device. The process is thereby loaded to the device upon initialisation of the application. For a pull request, the destination of the components to be migrated is the device the request was made from. The pull activity is activated upon a user request, since there are no guard expressions in the initial arrow.

The “detect” condition on the arrow indicates that the connector should only deploy this application if it has not been deployed before. This is done because the developer considers the impact of the local resources on the output of the pull activity. Alternatively the developer could specify that the component is always delivered, or that the component replaces any existing ones on the device. The developer can also use constructs for specifying whether to move or copy the behaviour or whether to migrate in a stateless or stateful manner.

As indicated by the *depends* arrow, the display component may be deployed with an editor component. Because the device may have limited memory capabilities, considering impact from device context, the device checks for memory required to determine if the editor functionality can be attached. The decision to load a component can be based on whether a component can be used remotely, likelihood of disconnections or other factors as well. The interaction model may impact upon deployment decisions, as interacting components may preferably be executed on the same device to avoid network access.

Based on this diagram the display activity will not execute as it is preceded by a *depends* arrow. The purpose of this diagram is not to specify the execution of the display activity, but to specify when it will be deployed on the device. The execution of the display activity is specified in the interaction aspect. The final state is not included for simplicity, and because it is ignored in the router generation algorithm presented in Section 4.1. The distribution and deployment processes will terminate when the interaction process terminates.

### *3.5 Data Distribution*

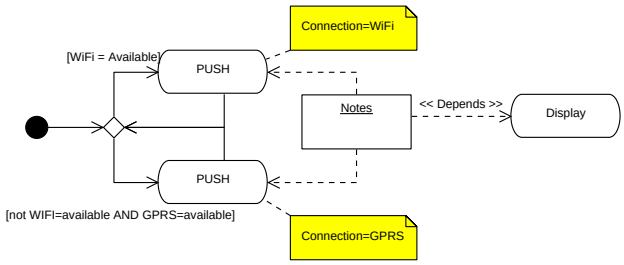


Fig. 4. Distribution Activity Diagram

In the data distribution view the developer must consider the deployment and interaction processes, and determine what data elements may need to be distributed over a network. For each required data element, a process, which describes how it is transferred, is created. This process includes any pre- or post- processing required (e.g. compression). The process description may be impacted upon by several factors such as the local device memory, available bandwidth, or available local processing to handle the incoming data. A data distribution event could occur when an interaction fails because there is no

local component which can handle a request for processing, or based on other events such as changes in location, or explicit requests for data.

Movement of data is specified similarly to movement of activities, by using push or pull activities. In our example process, two channels are specified to push the notes to the mobile device, either via GPRS or via WiFi. On the train the user has access to WiFi, while on the taxi only GPRS is available. The distribution of the notes should use any available channel, but preferably WiFi. In this process we assume that the same data is sent by both connectors. However, pre- processing could be performed so that less data, or compressed data is sent if the bandwidth is lower. In this case a pre-processing activity should be inserted before using the GPRS channel.

In the data aspect, data can be specified to have dependencies to other data elements, or to behaviour. These elements are then also transmitted with the data. For example in our process, display behaviour is required to view the notes. The activation of the deployment process for the display activity is then done through data distribution. This will cause a push of the display component without an explicit user request. The dependencies of the display component will be analysed to see if the editor is also delivered. This is an example of how the data distribution aspect can impact upon deployment.

The destination of a data transmission is specified as a set of properties (using UML comments) which the destination device/entity must hold. During execution these properties are attached to the object as metadata, and connectors with those properties will take these data objects.

Because of the impact of disconnection upon data transmission, the specification in Figure 4 uses multiple channels. The process is then started when the user starts the application. This process executes on the server side as follows. Initially, if there are notes available, and WiFi is available, then this connector is activated, the notes are read by the connector and transmission starts. If a failure occurs, the connector stops and outputs the remaining data to the Space. If the GPRS is now available, it becomes activated, and starts sending the remaining data. If this channel fails, then again the remaining data is written to the Space and is read by the first channel to become available. When there is no more data, the process completes. In this example, activation specification is impacted upon by availability of channels. Alternatively bandwidth or changes to user location can be used to determine when to send data to the device.

Any data sent during a disconnection is automatically buffered in the Space. A store and forward architecture [9] is therefore automatically achieved, and need no extra programming effort. In the example, the GPRS component will send data until it fails, however, it should be possible to interrupt this connector

if the WiFi becomes available. This would require the GPRS component to abort its activity under certain conditions. This is currently not supported.

The distribution of data, and deployment of behaviour may be difficult and time consuming to specify properly because of the impact of resource constraints. To enable easier application development, and to avoid the complexities of deployment or data distribution, it should be possible for the developer to use certain default views. These default specifications should specify standard rules applying to all data and components which needs to be distributed.

## 4 From process-based to event-based models

This section focuses on the issue of generating coordinators for process execution from UML activity diagrams. We first describe the technique for generating coordinators from UML activity diagram restricted to control-flow constructs. We then show how data-flow aspects are incorporated. Because all models are described very similarly, and distribution and deployment follow the same protocols as interaction, the algorithm for creating event-based models is applicable to all aspects.

### *4.1 Translating control flow*

For each action in the activity diagram, either a component or a connector (in case it is an external application) is created. A number of routers are also generated for each action. The input sets for these routers are generated according to the algorithm sketched using a functional programming notation in Figure 5 and explained below. The main function defined by this algorithm (namely AllInputSets) takes as input an activity diagram represented as a set of nodes (action, decision, merge, fork, join, initial, and final nodes) inter-linked through edges. From there, it generates a set of input sets (see definition of input set in Section 2.2). The input sets produced by this algorithm can then be used to create a collection of routers (one router per input set) that collectively are able to coordinate the execution of instances of the process in question. Intuitively, each input set encodes one possible way of arriving to a given node in the activity diagram.

#### *4.1.1 Algorithm for input sets generation*

The algorithm focuses on a core subset of activity diagrams covering only initial and final nodes, action nodes, and control nodes (i.e. decision, merge, fork,

and join nodes) connected by edges. In particular, the algorithm does not take into account object flow (which is discussed later). Without loss of generality, the algorithm assumes that all conditional guards in the activity diagram are specified in disjunctive normal form. Also without loss of generality, the algorithm assumes that there are no "implicit" forks and joins in the diagram. An implicit fork (join) occurs when several edges leave from (arrive to) an action node. In this case, the semantics of this fragment of the diagram is the same as that of a diagram in which this action node only has one outgoing (incoming) edge leading to (originating from) a fork node (a join node). Thus implicit forks and joins should be replaced by explicit fork and join nodes prior to applying this algorithm.

```
AllInputSets(p: Process) :
        let {x_1, ..., x_n} = ActionNodes(p) in
            InputSets(x_1) ∪ ... ∪ InputSets(x_n)
InputSets(x : Node) :
    let {t_1, ... t_n} = IncomingEdge(x) in
        return InputSetEdge(t_1) ∪ ... InputSetEdge(t_n)

InputSetsEdge(e : Edge) :
    let x = Source(t)
        if NodeType(x) = "action"
          return CompletionObject(x)
        else if NodeType(x) = "initial"
                return ProcessInstantiationObject(Process(x))
        else if NodeType(x) ∈ {"decision", "fork"}
                let {c_1, ..., c_n} = Disjuncts(Guard(t)),
                    {i_1, ..., i_n} = InputSets(Source(t)) in
                    return {{c_1} ∪ i_1, ..., {c_1} ∪ i_n},
                                ...
                                {c_n} ∪ i_1, ..., {c_n} ∪ i_n}
        else if NodeType(x) = "merge"
                let {t_1, ..., t_n} = IncomingEdge(x) in
                    return InputSetsEdge(t_1) ∪ ... ∪ InputSetsEdge(t_n)
        else if NodeType(x) = "join"
                let {t_1, ..., t_n} = IncomingEdge(x),
                    {⟨ i_1,1, ..., i_1,n⟩,
                     ...
                     ⟨ i_m,1, ..., i_m,n⟩} =
                    InputSetsEdge(t_1) × ... × InputSetsEdge(t_n) in
                    return {i_1,1 ∪ ... ∪ i_1,n,
                                ...
                                i_m,1 ∪ ... ∪ i_m,n}
```

Fig. 5. Algorithm for deriving input sets from an activity diagram.

Figure 5 defines three functions: the first one, namely AllInputSets generates all

the input sets for a process by relying on a second function, namely InputSets, which generates a set of input sets for a given node of the diagram. This latter function relies on a third (auxiliary) function named InputSetsEdge, which produces the same type of output as InputSets but takes as parameter an edge rather than a set. This definition of InputSetsEdge operates based on the node type of the source of the edge, which may be an action node, an initial node, or one of the four types of control nodes. If the edge's source is an action node, a single input set is returned containing a completion object (see Section 2.2) for that action. Intuitively, this means that the edge in question may be taken when a completion object corresponding to that action is placed on the Space. Similarly, if the source of the edge is the initial node of the activity diagram, a single input set with a "process instantiation" object is created, indicating that the edge in question will be taken when an object is placed on the Space signalling that a new instance of the process must be started. If the edge's source is a control node, the algorithm keeps working backwards through the diagram, traversing other control nodes, until reaching action nodes. In the case of an edge originating from a decision or a fork node, which is generally labeled by a guard (or an implicit "true" guard if no guard is explicitly given), the edge's guard is decomposed into its disjuncts, and an input set is created for each of these guards. This is done because the elements of an input set are linked by an "and" (not an "or") and thus an input set can only capture a conjunction of elementary conditions and completion/instantiation objects (i.e. a disjunct). Finally, in the case of an edge originating from a "merge" (resp. a "join"), the function is recursively called for each of the edges leading to this merge node (join node), and the resulting sets of input sets are combined to capture the fact that when any (all) of these edges is (are) taken, the corresponding merge node (join node) may fire.

The following notations are used in the algorithm:

- ActionNodes(p) is the set of action nodes contained in process p (described as an activity diagram).
- Source(e) is the source activity of edge e
- Guard(e) is the guard on edge e
- Disjuncts(c) is the set of disjuncts composing condition c
- IncomingEdge(x) is the set of edges whose target is node x
- NodeType(x) is the type of node x (e.g. "action", "decision", "merge", etc.)
- Process(x) is the process to which node x belongs.

### *4.1.2 Example*

Figure 6 describes the router for the "CheckTraffic" encoded in XML syntax to facilitate transfer of design data to the runtime system. XML is used because

of its generality, however the concept of router is not restricted to XML. This action node will only have one router associated to it because there is only one path leading to the execution of this action. Indeed, to execute this action, it is necessary that both the "check presentation time" and the "check train availability" actions have completed, and in addition that the condition "not ontime and train" evaluates to true, and this condition does not contain any disjunction. When all these conditions are satisfied, the router will produce an enabling object that will eventually be picked up by the connector associated to action "check traffic".

```
<Router name = ‘‘CheckTrafficEnabler’’>
  <Input>
   <Template>
   <CompletionObject actionName=’’CheckPresentationTime’’ piid=’’var:X’’/>
   </Template>
   <Template>
  <CompletionObject actionName=’’CheckTrainAvailability’’ piid=’’var:X’’/>
   </Template>
   <Condition>
    <Equality variable=’’ontime’’ value=’’false’’/>
   </Condition>
   <Condition>
    <Equality variable=’’train’’ value=’’true’’/>
   </Condition>
  </Input>
  <Output>
   <EnablingObject action=’’CheckTraffic’’ piid=’’var:X’’/>
  </Output>
</Router>
```

Fig. 6. Sample router

It can be noted in this example that the process instance identifier (piid) attribute of the completion object templates are associated with a variable. In XML syntax, an XML namespace (aliased "var") is reserved to refer to variables. The AOS is capable of interpreting collections of object templates where some of the attributes are associated with such variables and to match these templates in a way that if the same variable is associated with attributes of two different templates, then the objects matching these templates should contain the same values for these attributes.

### *4.2 Incorporating data-flow*

Data flow (or more precisely *object flow*) in activity diagrams is represented by object nodes, represented as rectangles. Object nodes are directly linked to a "producing" action preceding the object node. They are also linked, either directly or through the intermediary of a number of control nodes, to one or several "consuming" action node(s) following the object node.

In terms of the proposed technique, object flows are treated as follows. The input into a component is used to specify the contents of the enablement mes-

sage from a router (its output set). This enablement message will contain a specification of the objects which the component should read before processing. The production of objects for a given object node is the responsibility of the component corresponding to the action node directly preceding this object node (i.e. the producing action). Output specifications are therefore ignored when creating routers. The destination of output objects is used to create a router which will look for objects of the type specified by the properties of the destination, and attach metadata to it, before writing it back to the Space. Because components and connectors themselves pull the required data of the Space. The required component or connector will find the data with properties it is looking for and read it from the Space. We assume that all guard expressions are specified outside of the data input arrows. The effects on the arrows are included in the metadata, and the connector will use this to specify deployment as required.

In the data distribution example, the notes are input into one of the push activities. The comments attached to the push activities are used to create a router which, upon seeing this data element, attaches the comments as metadata to the data object. The connectors can thereby read the data containing the required element. In case of multiple destinations, no metadata is attached, and reading will be determined either non-deterministically, or as in this case, through control flow.

### *4.3 Inserting or Changing coordinators*

By inserting or changing coordinators, users, administrators and/or developers can re-route data and control in an already deployed composite application in order to steer it into executions paths not foreseen in the original process model, thereby facilitating the personalisation and adaptation of these applications. The developer can design additional processes to cater for new requirements and deploy these into the event-based execution space without having to restart the existing application. Also in certain situations, some functionality may or should be made unavailable. A context change may mean that some processing can not be performed, or a user moving outside a firewall may prevent her from executing certain applications. In our example, it may happen that the system takes too much time to contact the other meeting participants to check if the meeting can be postponed (i.e. the execution of the "postpone meeting" may take more time than the user is willing to wait for). In this case, a user may indicate that she does not wish to be delayed by this action, but instead, if the "Check Traffic" action is completed and if the traffic conditions are OK, she would immediately take a taxi. This adaptation can be achieved by activating the router specified in a concrete XML syntax in Figure 7. In this XML fragment, we assume that the piid of the process instance

for which this modification is to be done is 1. The element `StopSet` indicates that this router is disabled if the "Postpone Meeting" action is completed. Thus this router will only place an enabling object to trigger action "Catch Taxi" if the action "Check Traffic" completes before "Postpone Meeting" and the corresponding boolean expression evaluates to true.

```
<Coordinator name = 'with participants''>
  <Input>
   <Template>
    <CompletionObject action=''CheckTraffic'' piid=''1''/>
   </Template>
   <Condition>
    <Equality variable=''traffic'' value=''OK''/>
   </Condition>
  </Input>
  <Output>
     <EnablingObject action=''CatchTaxi'' piid=''1''/>
  </Output>
  <StopSet>
    <CompletionObject action=''PostponeMeeting'' piid=''1''/>
  </StopSet>
</Coordinator>
```

Fig. 7. Sample router for process adaptation

The above adaptation could arguably be achieved by modifying the process model. However, in this case, significant tool support would be required and model versioning may become an issue. In contrast, enabling an event-based rule (encoded as a router) provides a more lightweight adaptation mechanism.

More radical changes may also be made. For example, consider a user that prefers taxis over trains in any case and so would always catch taxis regardless of traffic conditions and amount of time before the meeting. In this case, a router may be introduced that enables the action "CatchTaxi" immediately upon process instantiation when the process instance is started by the user in question. At the same time, all other routers for that process instance would be disabled, except the ones for download notes and display notes. Other possibilities also exist, such as the user wanting to choose the means of transport with the best network access. For example, the user may prefer the train, if it means having access to WiFi.

Specifies "dynamic" changes to composite applications may be achieved, for example, by means of personalisation applications running as active objects and disabling or enabling routers or placing completion or enabling objects according to an adaptation logic previously coded by a developer. Another option is to provide users with options for adapting/personalising applications. When a user manually selects one of these options, a number of coordinators are enabled and/or completion and enabling objects are written to or taken off the Space. Of course, this mechanism may be abused and lead to undesirable effects such as deadlocks. However, as shown above, adaptation may be scoped to specific process instances to avoid affecting a wider user base. In addition, as certain adaptations become permanent, they may be propagated back to

the process model resulting in a new process model being deployed.

## 5 Related Work

Previous work in mobile computing, has focused on building architectures to enable adaptation rather than methodologies. Existing mobile methodologies, typically present either high-level business analysis, or component internals such as algorithms or user interfaces. In contrast we focus on component externals, meaning a specification of component interaction, and deployment, as well as data distribution.

Process-oriented application development has been the subject of significant attention in the last decade, prompting the emergence of a large number of process modeling and execution languages. However, the platforms supporting these languages adopt an approach to process-oriented application development that is not suitable in scenarios where personalisation and adaptation are prominent requirements. These platforms typically rely on the static definition of process models and allow little change to occur without redeployment.

Proposals in the area of adaptive and flexible workflow [14] generally focus either on a priori adaptation (e.g. attaching exception handling policies to a process model) or on dealing with changes in the process model. In contrast, our proposal shows that if an event-based coordination model is used at the execution layer, it is possible to make fine-grained changes to specific parts of the process and to confine these changes to specific process instances, without altering the process model. In other words, the process model can be used as a reference to deal with the majority of cases but deviations can occur for specific cases based on the activation or de-activation of the rules composing the event model.

Parallels can be drawn between our approach and the one followed in case handling systems [2] where human workers route cases (i.e. process instances) manually based on information associated to each case and contextual information such as workload and resource availability. However, case handling is targeted at processes composed mostly of manual tasks. In contrast, our proposal is targeted at processes in which tasks are delegated to software applications so that it is not possible to count on human intervention at each step of the process.

There exist a large body of proposals in the area of coordination architectures, and in particular Space-based ones. Some of these architectures (e.g. Mars [4] and Limone [7]) support the definition of reaction rules to coordinate application components, similar to the way coordinators operate in our

framework. However, despite their potential synergies, proposals in the areas of coordination architectures on the one hand, and process-oriented application development on the other, have so far evolved independently – a notable exception being the work by Tolksdorf [17] who describes a Space-based architecture for routing XML documents through processing steps encoded in XSL. A major novelty of our proposal is that it seamlessly combines techniques from coordination-based and from process-oriented software architectures.

This paper partly builds upon previous work on decentralised orchestration of process-oriented composite services specified as UML statecharts [3]. In this prior work, an algorithm was proposed that bears some similarities with the one presented in Figure 5. In addition to technical differences between the algorithms, stemming in part from the use of activity diagrams (version 2.0) rather than statecharts, the proposal of this paper differs from the previous one in the use that it makes of the output of the algorithm: Instead of using this output for decentralised orchestration, it uses it for event-based centralised orchestration based on coordination middleware.

The proposal in this paper can also be seen as a refinement of the architecture presented in [15], where agents and tuple Spaces are combined in an architecture for service composition. In the present paper, we have presented a concrete approach to encode and execute event-based models and we have detailed a method for generating event-based models from process-based ones. By encoding event-based models as active objects it is possible to achieve various forms of adaptation.

## 6 Conclusion and Future Work

This paper has shown that dynamic aspects of mobile applications can be specified using process-models, which can be translated into event-based models for more flexible execution. This method supports just-in-case adaptation by allowing developers to specify adaptation in an activity diagram, and supports just-in-time adaptation as new behaviour (and behaviour specifications) can be overlayed on top of existing behaviour to adapt to specific runtime requirements.

This paper has covered some issues in how UML can be applied to adaptable mobile application design, in the context of a mobile process example. Our analysis of design dependencies for mobile applications are being further developed and refined. Several other issues are of importance to address in our methodology, such as database access and synchronization. Such aspects may require additions to the UML constructs. We aim to specify completely the dependencies, and develop a toolkit to aid the developer in navigating the as-

pects, sub-aspects and their dependencies. The toolkit will provide developers with design suggestions and outline the impacts of these on the environment and the application.

Another possible direction for future work is to design a mapping from event-based models to process models. The idea would be to automatically derive a process model from a collection of routers. This "reverse" mapping would assist developers in propagating changes in the event-based model to the process model, when it is decided that these changes should be made permanent. Techniques such as those developed in the setting of process mining, could provide insights for designing this reverse mapping.

**Acknowledgments** The first author is funded by an SAP-sponsored scholarship. The second author is funded by a Queensland Government Smart State Fellowship co-sponsored by SAP.

## References

[1] W. M.P. van der Aalst. How to handle dynamic change and capture management information: An approach based on generic workflow models. *Computer Systems Science and Engineering*, 15(5):295–318, 2001.

[2] W.M.P. van der Aalst, M. Weske, and D. Grünbauer. Case handling: A new paradigm for business process support. *Data and Knowledge Engineering*, 53(2):129–162, 2005.

[3] B. Benatallah, M. Dumas, and Q.Z. Sheng. Facilitating the rapid development and scalable orchestration of composite web services. *Distributed and Parallel Databases*, 15(1):5–37, January 2005.

[4] G. Cabri, L. Leonardi, and F. Zambonelli. Reactive tuple spaces for mobile agent coordination. In *Proceedings of the Second International Workshop on Mobile Agents (1998)*, pages 237–248, Stuttgart, Germany, 1999. Springer Verlag.

[5] K. Elms, S. Milliner, and J. Vayssiere. Object spaces with active objects. U.S. Patent Application # 2004P00851US, filed 29 December 2004.

[6] T. Fjellheim, S. Milliner, M. Dumas. Middleware Support for Mobile Applications. *International Journal of Pervasive Computing and Communications*, 1(2): 75-88, June 2005.

[7] C-L. Fok, G-C. Roman, and G. Hackmann. A lightweight coordination middleware for mobile computing. In *Proceedings of the 6th International Conference on Coordination Models and Languages*, pages 135–151, Pisa, Italy, February 2004. Springer Verlag.

[8] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming*, 2(1):80–112, January 1985.

[9] R. Grimm System support for pervasive applications. Ph.d. Thesis. University of Washington. 2002. http://www.cs.nyu.edu/rgrimm/one.world/papers.html

[10] S-Y. Hwang and Y-F. Chen. Personal workflows: Modeling and management. In *Proceedings of the 4th International Conference on Mobile Data Management*, 2003.

[11] O. Lehmann, M. Bauer, C. Becker, D. Nicklas. From Home to World - Supporting Context-aware Applications through World Models, Second IEEE International Conference on Pervasive Computing and Communications. March 2004. IEEE Computer Society.

[12] D. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.* Addison-Wesley Professional, 2002.

[13] P.J. McCann, and G-C. Roman. Compositional Programming Abstractions for Mobile Computing. *IEEE Transactions on Software Engineering*, 24(2):97–110, 1998.

[14] S. Rinderle, M. Reichert, and P. Dadam. Correctness criteria for dynamic changes in workflow systems - a survey. *Data and Knowledge Engineering*, 50(1):9–34, 2004.

[15] Q.Z. Sheng, B. Benatallah, Z. Maamar, M. Dumas, and A.H.H. Ngu. Enabling personalized composition and adaptive provisioning of web services. In *Proceedings of the International Conference on Advanced Intormation Systems Engineering*, pages 322–337, Riga, Latvia, June 2004. Springer Verlag.

[16] José Pascual Molina Massó and Jean Vanderdonckt and Pascual González López. Direct manipulation of user interfaces for migration. In *Proceedings of the 11th international conference on Intelligent user interfaces*, Sydney, Australia, 2006. ACM Press.

[17] R. Tolksdorf. Coordination technology for workflows on the web: Workspaces. In *Proceedings of the 4th International Conference on Coordination Models and Languages*, pages 36–50, Limassol, Cyprus, September 2000. Springer Verlag.

[18] P. Wohed, W. M.P. van der Aalst, M. Dumas, A. H.M. ter Hofstede, and N. Russell. Pattern-based Analysis of the Control-flow Perspective of UML Activity Diagrams. In *Proceedings of the International Conference on Conceptual Modelling (ER)*, Klagenfurt, Austria, October 2004. Springer Verlag.